**Comparison of Linear System Solvers Applied
to Diffusion-Type Finite Element Equations**

by

Anne Greenbaum, Congming Li, Han Zheng Chao

Ultracomputer Note #126

September, 1987

**Comparison of Linear System Solvers Applied
to Diffusion-Type Finite Element Equations**

by

Anne Greenbaum, Congming Li, Han Zheng Chao

Ultracomputer Note #126

September, 1987

This work was supported in part by the Applied Mathematical Sciences Program of the US Department of Energy under contract DE-AC02-76ER03077 and in part by contract number W-7405-ENG-48.

ABSTRACT

Various iterative methods for solving the linear systems associated with finite element approximations to self-adjoint elliptic differential operators are compared based on their performance on serial and parallel machines. The methods studied are all preconditioned conjugate gradient methods, differing only in the choice of preconditioner. The preconditioners considered arise from diagonal scaling, incomplete Cholesky decomposition, hierarchical basis functions, and a Neumann-Dirichlet domain decomposition technique. The hierarchical basis function idea is shown to be especially effective on both serial and parallel architectures.

1. Introduction

The preconditioned conjugate gradient method [9,4] has proved to be an effective iterative procedure for solving the linear systems arising from finite element and finite difference approximations to self-adjoint elliptic partial differential equations. Recent effort has focussed on developing efficient preconditioners for such problems, to insure rapid convergence of the iteration.

In use for some time have been diagonal scaling [4] and incomplete Cholesky [12] preconditioners. The application of these preconditioners requires knowledge only of the linear system to be solved, not of the partial differential equation or differencing technique used. When applied to problems arising from the discretization of a partial differential operator on some mesh, however, these methods slow down considerably as the mesh spacing becomes small.

In recent years, preconditioners have been developed that insure convergence of the conjugate gradient method at a rate that is independent of, or only mildly dependent on, the mesh spacing. Such preconditioners usually require some knowledge of the partial differential equation and/or the mesh and differencing technique used, and so they are less general than diagonal scaling and incomplete Cholesky decomposition. For fine enough meshes, however, they are sure to outperform these other preconditioners in terms of number of iterations required to achieve convergence. Two such preconditioning ideas are considered here. One arises from using hierarchical basis functions [16], instead of the usual nodal basis functions, to represent the finite element space. The other is an iterative substructuring technique known as Neumann-Dirichlet domain decomposition [6], which requires dividing the domain of the problem into pieces and at each iteration solving separately on the different pieces. While it can be shown that for a wide class of finite element approximations to self-adjoint elliptic partial differential equations the number of iterations required by these methods is bounded by $O(\log h^{-1})$, the constant multiplying the $\log h^{-1}$ factor is problem dependent, and so numerical experiments are needed to determine the suitability of the methods for solving particular problem types.

All of these methods offer some opportunity for exploiting parallelism. In fact, for a fixed number of processors P , if the problem size is large enough, then each of these methods can achieve a full factor of P speedup over its serial execution time. The only differences lie in the size and serial execution time of problem needed to achieve this factor of P speedup; i.e., in the "granularity" of parallel tasks compared to the total work performed.

In the following section the algorithms are briefly described, with references to the relevant literature for proofs of convergence rates, etc. Next the different parts of each method are analyzed with regard to number of operations required and amount of parallelism present. Asymptotic estimates of serial and parallel work are given. Finally, results of numerical tests performed on the NYU Ultracomputer Prototype are reported.

2. The Algorithms.

As mentioned previously, the algorithms described here are all preconditioned conjugate gradient methods [9,4], differing only in the choice of preconditioner. When the conjugate gradient method is employed to solve a symmetric positive definite linear system $A\bar{x} = b$, its convergence rate can be estimated in terms of the square root of the condition number of the coefficient matrix A . If the matrix A is well-conditioned, convergence will be rapid. If A is not well-conditioned, one often uses a "preconditioning" matrix M , having the property that (1) linear systems with coefficient matrix M are relatively easy to solve, and (2) the matrix $M^{-1}A$ is well-conditioned. One then solves a linear system with coefficient matrix M at each iteration of the conjugate gradient algorithm, and the number of iterations required depends on the square root of the condition number of $M^{-1}A$. The conjugate gradient algorithm with preconditioner M can be stated as follows:

Given an initial guess x^0 , compute $r^0 = b - Ax^0$,
and solve for z^0 the linear system $Mz^0 = r^0$.
Set $p^0 = z^0$. Compute $\hat{a}_0 = z^{0T}r^0$. For $k=1,2,\dots$,
Compute Ap^{k-1} .
Set $a_{k-1} = \frac{\hat{a}_{k-1}}{p^{k-1T}Ap^{k-1}}$.
Take $x^k = x^{k-1} + a_{k-1}p^{k-1}$.
Compute $r^k = r^{k-1} - a_{k-1}Ap^{k-1}$.
Solve for z^k the linear system $Mz^k = r^k$.
Check for convergence. If convergence criterion satisfied, then quit. Else,
Set $\hat{a}_k = z^{kT}r^k$ and $b_k = \frac{\hat{a}_k}{\hat{a}_{k-1}}$.
Compute $p^k = z^k + b_kp^{k-1}$.

The simplest preconditioner to use in the conjugate gradient algorithm is the diagonal of A , $M = \text{diag}(A)$. Of all diagonal preconditioning matrices, it can be shown [1] that this is nearly optimal for minimizing the condition number of $M^{-1}A$. We will refer to the conjugate gradient method with this diagonal scaling preconditioner as DSCG (diagonally scaled conjugate gradients).

A more sophisticated preconditioner is obtained by performing an "incomplete Cholesky factorization" [12] of the matrix A . Here one factors the matrix A approximately into the product of a lower triangular matrix and its transpose, where the triangular factors have nonzero entries only in positions where the corresponding triangular parts of A have nonzeros. The nonzero entries are chosen so that the product of the incomplete Cholesky factors matches A in places where A has nonzeros, but this product also has some nonzero elements in places where A has zeros. The form of the incomplete Cholesky factors and their product, for a nine-striped matrix A , such as that arising from a piecewise bilinear finite element approximation, is pictured below:

The conjugate gradient algorithm with the incomplete Cholesky preconditioner will be referred to as ICCG (incomplete Cholesky conjugate gradients).

To describe the other two preconditioners, we must first consider the nature of a finite element approximation for a self-adjoint elliptic partial differential equation

$$Lu = f$$

defined on a domain Ω , and satisfying boundary conditions, say,

$$u = 0 \text{ on } \Gamma_1, \quad u_n = 0 \text{ on } \Gamma_2,$$

$$u_n = \text{normal derivative of } u, \quad \Gamma_1 \cup \Gamma_2 = \Gamma = \text{boundary of } \Omega.$$

An approximate solution $\hat{u}$ for such a problem can be chosen from a finite dimensional space S_n in such a way that the "energy" norm of the error

$$\langle L(u-\hat{u}), u-\hat{u} \rangle$$

is minimized, or, equivalently, in such a way that the residual is orthogonal to every element of S_n :

$$\langle L(u-\hat{u}), \hat{v} \rangle = 0, \quad \forall \hat{v} \in S_n.$$

[Here the symbol $\langle \cdot, \cdot \rangle$ denotes the usual L_2 inner product, after integrating by parts or applying Green's theorem to eliminate high order derivatives.]

To find this approximate solution, one uses a "basis" $\phi_1, \phi_2, \dots, \phi_n$ for the space S_n and writes $\hat{u}$ as a linear combination of the basis functions:

$$\hat{u}(x) = \sum_{j=1}^n \alpha_j \phi_j(x).$$

The coefficients $\alpha_1, \dots, \alpha_n$ are chosen so that $\hat{u}$ satisfies

$$\langle L(u-\hat{u}), \phi_i \rangle = \langle f, \phi_i \rangle - \sum_{j=1}^n \alpha_j \langle L\phi_j, \phi_i \rangle = 0, \quad i=1, \dots, n,$$

or, in matrix form,

$$A \bar{\alpha} = \bar{f}, \quad \bar{\alpha} = \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix}, \quad \bar{f} = \begin{pmatrix} \langle f, \phi_1 \rangle \\ \vdots \\ \langle f, \phi_n \rangle \end{pmatrix},$$

$$A_{ij} = \langle L\phi_j, \phi_i \rangle, \quad i, j=1, \dots, n.$$

Ordinarily, the basis functions $\phi_1, \dots, \phi_n$ are chosen to be nonzero over just a few intervals of the mesh. The matrix A is then quite sparse, having zero entries in positions i, j such that ϕ_i and ϕ_j are nonzero over disjoint intervals. For example, in one dimension, a typical choice for S_n is the space of continuous piecewise linear functions satisfying the essential boundary conditions, and the basis functions $\phi_1, \dots, \phi_n$ are taken to be one at one node and zero at all others, as pictured below:

In two dimensions, it is common to use triangular elements with piecewise linear approximation functions, or rectangular elements and piecewise bilinear approximations. The usual "nodal" basis functions are again nonzero at one node and zero at all others.

When the matrix A is formed from the energy inner products of such basis functions and the operator L is a second-order elliptic differential operator with appropriate boundary conditions, it can be shown that the condition number of the resulting linear system is $O(h^{-2})$, where h is the minimum mesh spacing of the regular grid. Thus, the conjugate gradient algorithm applied to such a linear system, with no preconditioner, requires $O(h^{-1})$ iterations to achieve a desired fixed level of accuracy.

The same finite element approximation can be obtained using different basis functions. If the usual nodal basis functions are replaced by "hierarchical" basis functions [16], it can be shown that, in two dimensions, the condition number of the resulting matrix $\tilde{A}$ is of order $O(\log h^{-1})^2$, and so the conjugate gradient algorithm can achieve a desired fixed level of accuracy in $O(\log h^{-1})$ iterations. Hierarchical basis functions consist of the usual nodal basis function(s) on a coarsest grid, together with the nodal basis functions for a finer grid corresponding to nodes that are not present in the coarsest grid, together with the nodal basis functions for a still finer grid corresponding to nodes that are not present in any of the coarser levels, etc., up to the finest grid on which it is desired to solve the problem. The hierarchical basis functions for the one-dimensional space of continuous piecewise linear functions are pictured below:

In two dimensions, with triangular or rectangular grids, the picture is similar.

The linear system arising from the hierarchical basis functions, $\psi_1, \dots, \psi_n$, is given by

$$\tilde{A}\tilde{\beta} = \tilde{f}, \quad \tilde{\beta} = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_n \end{bmatrix}, \quad \tilde{f} = \begin{bmatrix} \langle f, \psi_1 \rangle \\ \vdots \\ \langle f, \psi_n \rangle \end{bmatrix},$$

$$\tilde{A}_{ij} = \langle L\psi_j, \psi_i \rangle, \quad i, j = 1, \dots, n,$$

where the approximate solution $\hat{u}(x)$ has been written in the form

$$\hat{u}(x) = \sum_{j=1}^n \beta_j \psi_j(x).$$

The matrix $\tilde{A}$ arising from the hierarchical basis is much less sparse than the matrix A associated with the nodal basis representation. Fortunately, however, it is not necessary to actually compute the matrix $\tilde{A}$. We need only be able to multiply $\tilde{A}$ times a vector in order to

solve the linear system using the conjugate gradient algorithm.

Let S be the linear transformation which maps coefficient vectors $\tilde{\beta}$ of the hierarchical basis representation of a function in S_n into the corresponding coefficient vector $\bar{\alpha}$ of the nodal basis representation of the function. Then it is not hard to see [16] that the matrix $\tilde{A}$ is given by

$$\tilde{A} = S^T A S ,$$

where A is the matrix associated with the nodal basis functions. Thus, to multiply $\tilde{A}$ times a vector, we first transform that vector to the nodal basis representation, then multiply by A , then perform the transpose of the transformation.

It is also easy to see that applying the unpreconditioned conjugate gradient algorithm to the matrix $\tilde{A} = S^T A S$ is equivalent to applying the preconditioned conjugate gradient algorithm to the matrix A , with preconditioner M satisfying

$$M^{-1} = S S^T .$$

By applying the algorithm in this form, we avoid explicitly translating from nodal to hierarchical bases in the beginning and translating back at the end. The conjugate gradient algorithm with this hierarchical basis preconditioner will be referred to as HBCG (hierarchical basis conjugate gradients).

Note that the preconditioner M in no way depends on the differential operator L ! It is entirely determined by the finite element space S_n and by the nodal and hierarchical basis functions for S_n . While Yserentant showed [16] that the condition number of $M^{-1}A$ is $O(\log h^{-1})^2$ for all reasonably smooth self-adjoint elliptic differential operators L , one would expect that if M is a good preconditioner for one operator it would not be a good preconditioner for a very different differential operator. This is borne out in the numerical examples of section 4. By considering diffusion-type operators with strongly varying diffusion coefficients, one can construct matrices for which this preconditioner is arbitrarily bad (though still dependent only logarithmically on the mesh size).

To avoid this problem, we used a diagonal scaling on the matrix $\tilde{A} = S^T A S$. According to Bauer's theorem [1], a nearly optimal diagonal preconditioner to use is $\tilde{D} = \text{diag}(S^T A S)$. This diagonal matrix requires additional time to compute, however, and we found that we obtained almost identical results using $D = \text{diag}(A)$ as the preconditioner for $\tilde{A}$. Thus, the unpreconditioned conjugate gradient algorithm was applied to the matrix

$$D^{-1/2} \tilde{A} D^{-1/2} , \quad D = \text{diag}(A) ,$$

or, equivalently, the preconditioned conjugate gradient method was used to solve the linear system $A\bar{x} = b$, with preconditioner M satisfying

$$M^{-1} = S D^{-1} S^T .$$

This algorithm will be referred to as HBDSCG (hierarchical basis diagonally scaled

conjugate gradients).

The final algorithm tried is an iterative substructuring technique known as Neumann-Dirichlet domain decomposition [6]. This idea requires dividing the domain of the problem into "boxes", as pictured below:

Some of the boxes are labelled "D" for Dirichlet subregions and some are labelled "N" for Neumann subregions. D-boxes may share a side only with N-boxes, and vice versa. Dirichlet points (mesh points in the interior of D-boxes) are numbered first, and Neumann points (mesh points in the interior and on the internal or free boundaries of N-boxes) are numbered last. With this ordering of unknowns, using the usual nodal basis representation of the finite element space, the matrix A takes the form

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{12}^T & A_{22} \end{pmatrix},$$

where the entries of A_{11} represent couplings between Dirichlet points, the entries of A_{22} couplings between Neumann points, and the entries of A_{12} couplings between Dirichlet and Neumann points.

Formally solving the linear system

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{12}^T & A_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

using Gaussian elimination, one writes

$$(A_{22} - A_{12}^T A_{11}^{-1} A_{12}) x_2 = b_2 - A_{12}^T A_{11}^{-1} b_1,$$

$$A_{11} x_1 = b_1 - A_{12} x_2,$$

and solves the Schur complement matrix,

$$C = A_{22} - A_{12}^T A_{11}^{-1} A_{12},$$

to obtain x_2 and then substitutes this into the original equation to obtain x_1 . The matrix C is usually dense and quite costly to compute and store. Fortunately, however, the matrix C

need not be computed in order to solve with the conjugate gradient algorithm. We need only be able to multiply C times a vector. This requires multiplication by the submatrices A_{12} , A_{12}^T , and A_{22} , and solving of a problem on the Dirichlet region. The D-boxes are uncoupled, and so A_{11} is a block diagonal matrix, with each block corresponding to one of the D-boxes of the domain. The individual blocks can be solved independently and in parallel.

Unfortunately, the matrix C is not well-conditioned, as argued in [3]. Hence a preconditioner is needed to insure rapid convergence of the conjugate gradient iteration. The preconditioner proposed in [6] is the matrix $A_{22}^{(N)}$ resulting from a finite element approximation (with the usual nodal basis functions) for a problem defined only on the Neumann part of the region:

$$(A_{22}^{(N)})_{ij} = \langle L\phi_j, \phi_i \rangle_{\Omega_2} \equiv \int_{\Omega_2} L\phi_j \cdot \phi_i \, dx.$$

It is shown in [14] that the condition number of the matrix

$$A_{22}^{(N)^{-1}} C$$

is of order $O(\log(\frac{d}{h}))^2$, where h is the minimum mesh width and d is the maximum N- or D-box width.

Different N-boxes couple at cross points. Hence, to solve $A_{22}^{(N)}$ we number these cross points last and write

$$A_{22}^{(N)} = \begin{pmatrix} B_{22} & B_{23} \\ B_{23}^T & B_{33} \end{pmatrix}$$

where B_{22} represents couplings between points of the Neumann region, other than cross points, and has a block diagonal structure with one block for each N-box of the region. To solve the linear system

$$\begin{pmatrix} B_{22} & B_{23} \\ B_{23}^T & B_{33} \end{pmatrix} \begin{pmatrix} z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} r_2 \\ r_3 \end{pmatrix},$$

we actually form the Schur complement

$$B = B_{33} - B_{23}^T B_{22}^{-1} B_{23}$$

and factor this banded matrix. At each iteration of the conjugate gradient algorithm, the factored matrix is then used to compute z_3 , and this is substituted into the original system to obtain z_2 by solving the block diagonal matrix B_{22} . The blocks of B_{22} can be solved independently and in parallel. This Neumann-Dirichlet preconditioner used with the conjugate gradient algorithm will be referred to as NDCG (Neumann-Dirichlet conjugate gradients).

3. Operation Counts and Asymptotic Estimates.

A single iteration of the preconditioned conjugate gradient algorithm given in section 2 requires the following operations:

- 1 matrix A times a vector (ATIMES)
- 1 solution of a linear system with coefficient matrix M (MSOLVE)
- 2 vector inner products (SDOT)
- 1 check for convergence, usually computing the 2-norm of a vector (SNRM2)
- 3 scalar times a vector plus a vector (SAXPY)

If A is a nine-stripped matrix, as in the examples of section 4, then the work per iteration, excluding the MSOLVE operation, is approximately

- 1 9n multiplies, 8n additions (ATIMES)
- 2 n multiplies, n additions (SDOT)
- 1 n multiplies, n additions (SNRM2)
- 3 n multiplies, n additions (SAXPY)

29n floating point operations (TOTAL)

With n processors, ignoring synchronization and communication time, the ATIMES and SAXPY operations could be performed in time $O(1)$, while the SDOT and SNRM2 operations would require time $O(\log n)$. For a fixed number of processors P , however, with $P \ll n$, the execution time of each of these operations will be approximately $1/P$ times its execution time on a single processor. Because of the processor synchronization required between operations, however, and because of the very simple nature of the operations performed, n may have to be very large compared to P in order to achieve this factor of P speedup.

Although the algorithms considered in this paper use very different sorts of MSOLVE operations, it will be argued later that for large enough n and a fixed number of processors P , each MSOLVE operation can also attain almost a full factor of P speedup, though some will require larger values of n than others. While a single iteration of the conjugate gradient algorithm can thus be parallelized, the different iteration steps must occur sequentially. One processor might begin the work of iteration step k while another is still working on step $k-1$,

but iteration k cannot be completed until after iteration $k-1$ is completed. Thus, the most important property of a preconditioned conjugate gradient method, for good performance on parallel machines, is that it requires few iterations to converge. Additional work in the MSOLVE operation at each iteration may be acceptable if it is parallelizable and if it significantly reduces the number of iterations required.

In the very simple DSCG method, M is taken to be $\text{diag}(A)$, and the MSOLVE operation requires n floating point divides (or n multiplies if $\text{diag}(A)^{-1}$ is precomputed). This operation is easily parallelized, and if n processors were used, the theoretical execution time would be $O(1)$. In practice, for a fixed number of processors P , if n is large enough, then a full factor of P speedup can be attained in this operation.

The MSOLVE operation in the ICCG method is more difficult to parallelize. This requires solving a sparse triangular linear system. If enough processors are available, $O(n^2)$, then this operation can theoretically be performed in time $O(\log^2 n)$ [13]. The algorithm for performing this banded triangular solve, however, requires more total operations than the usual method of backsolving and so is not practical when the number of processors is small compared to n . The standard method of backsolving for a sparse triangular linear system, such as that arising from a nine-point finite element approximation, can be partially parallelized. If the domain of the problem is a rectangle and if the natural ordering of nodes is used, then solution components can be computed in the order pictured below:

Values associated with the nodes along each (semi-) diagonal of the rectangle can be solved for simultaneously. If $\sqrt{n}/2$ processors are available, then the theoretical execution time for this backsolve is the number of diagonals, $O(\sqrt{n})$. If the number of processors P is much less than $\sqrt{n}/2$, then it can be shown as in [8] that almost a full factor of P speedup can be attained in this operation.

The MSOLVE operation in the HBCG and HBDSCG methods requires translating from the hierarchical basis functions to the nodal basis functions. This operation can be performed recursively, as described in [16]. First, nodal basis coefficients are computed at the nodes of the coarsest grid. Values at the nodes of the next finer grid are then taken as averages of the already computed surrounding nodal values. On the next finer grid, coefficients are again computed as averages of the surrounding nodal values, which have already been computed, etc. The transpose of this operation is performed similarly, only now one starts with the nodes that belong to the finest grid only and works down to the nodes of the coarsest grid. If n processors are available, then the time to perform this translation is the number of grid levels, $O(\log n)$. If the number of processors P is much less than n , then although not all processors can be kept busy all of the time, the percentage of time spent on grids with fewer than P points is small, and almost a full factor of P speedup can be attained in this operation, as well. It should be noted that the process of translating from hierarchical to nodal basis functions involves recursively working on each grid level, much as in the multigrid method. There is less work to do on each grid, however, as there is no relaxation step but only an interpolation-like operation to perform. Thus, a single iteration of the hierarchical basis conjugate gradient method should parallelize somewhat more efficiently than a single multigrid cycle. The disadvantage is that the hierarchical basis method requires more iterations (cycles) than the multigrid method — $O(\log h^{-1})$ as opposed to $O(1)$.

The final algorithm considered, NDCG, uses an MSOLVE routine that requires: (1) solving a banded linear system with coefficient matrix $B = B_{33} - B_{23}^T B_{22}^{-1} B_{23}$ of order equal to the number of crosspoints between N-boxes and of half-bandwidth equal to the number of crosspoints in a row of the grid, (2) multiplying by the matrix B_{23} , which couples Neumann points other than crosspoints to the crosspoints, and (3) solving the block diagonal matrix B_{22} , where each block represents a Neumann problem on one of the N-boxes. Because the conjugate gradient algorithm is applied to the matrix $C = A_{22} - A_{12}^T A_{11}^{-1} A_{12}$ in this case, rather than to the matrix A , the "ATIMES" routine is also different from that used with the other methods. It requires multiplying by the submatrices A_{12} , A_{12}^T , and A_{22} and solving the block diagonal matrix A_{11} , where each block represents a Dirichlet problem on one of the D-boxes.

Depending on the nature of the problem being solved, the MSOLVE and ATIMES operations might be carried out in several different ways. If the differential operator is equal to a constant times the Laplacian on each box and if the grid is uniform, then fast Poisson solvers can be used on the Dirichlet and Neumann subregions (though it is necessary to use some sort of embedding or Lagrange multiplier technique [15] on the Neumann subregions, since they do not contain the corner crosspoints). The work required for the fast solvers is approximately $10 m \log_2 m$ floating point operations, where $m \approx (d/h)^2$ is the number of points in a single D- or N-box.

The assumption of a Laplace operator on each box was unsuitable for some of the problems we considered, and so we solved the equations on each box using Gaussian elimination and a band matrix solver. Each diagonal block of A_{11} and B_{22} was factored prior to starting the conjugate gradient iteration, and then at each iteration the triangular factors were solved. The number of floating point operations required for each backsolve is approximately $4m^{3/2}$. The total work involved in the MSOLVE operation is then:

Solve band matrix B: $4d^{-3}$

Multiply by B_{23} : $12d^{-2}$

Solve block diagonal matrix B_{22} : $2n(d/h)$.

The ATIMES operation for NDCG requires:

Solve block diagonal matrix A_{11} : $2n(d/h)$

Multiply by A_{12} and A_{12}^T : $24\sqrt{n/d}$

Multiply by A_{22} : $17n/2$.

Note that if d/h is kept fixed as $h \rightarrow 0$, then the order and bandwidth of the matrix B differ by only a constant factor from those of the original matrix A. Hence, factoring B before the iteration requires $O(n^2)$ operations, just as are required to factor the matrix A directly. At each iteration, the backsolve on the banded triangular factors requires $O(n^{3/2})$ operations. On the other hand, if d is kept fixed as $h \rightarrow 0$, then the order and bandwidth of the blocks associated with each D- and N-box differ by only a constant factor from those of A. Hence, again, $O(n^2)$ operations are required to factor the blocks and $O(n^{3/2})$ operations must be performed at each iteration to backsolve the banded triangular factors. Thus, asymptotically, the operation count for the NDCG method is of the same order as that for direct Gaussian elimination, unless fast solvers or some other special solution technique can be used on the subproblems.

Table 1 summarizes the estimated iteration counts, work per iteration, and parallelism present in the implementations used for these different iterative methods.

Method	# Iterations	work/iteration	time/iteration with $O(n)$ processors	time/iteration with $P \ll n$ processors
DSCG	$O(h^{-1})$	$30 n$	$O(\log n)$	$\frac{t_{\text{serial}}}{P}$
ICCG	$O(h^{-1})$	$46 n$	$O(\sqrt{n})$	$P \ll \sqrt{n} / 2: \frac{t_{\text{serial}}}{P}$
	$O(\log h^{-1})$	$45 n$	$O(\log n)$	$\frac{t_{\text{serial}}}{P}$
HBDSCG	$O(\log h^{-1})$	$46 n$	$O(\log n)$	$\frac{t_{\text{serial}}}{P}$
NDCG	$O(\log \frac{d}{h})$	d/h fixed, $h \rightarrow 0$: $4(d/h)^3 n^{3/2}$ d fixed, $h \rightarrow 0$: $4dn^{3/2}$	time for banded triangular solve with bandwidth $O(\sqrt{n})$, size $O(n)$	$P \leq \frac{1}{2d^2}: \frac{t_{\text{serial}}}{P}$

Table 1. Work and Parallelism in Different Iterative Methods.

4. Numerical Results.

The problem considered is a time-independent version of the diffusion equation defined on the unit square with Dirichlet boundary conditions:

$$\nabla \cdot \rho \nabla u = f \quad \text{on } (0,1) \times (0,1)$$

$$u(x,0) = u(x,1) = u(0,y) = u(1,y) = 0.$$

Expressions for the diffusion coefficient $\rho(x,y)$ ranged from a constant to a mildly varying continuous function to a strongly varying piecewise constant. The right-hand side $f(x,y)$ was taken to be

$$f(x,y) = -2y(1-y) - 2x(1-x),$$

so that for $\rho=1$ the solution was $u(x,y)=x(1-x)y(1-y)$. The methods described previously were used to solve the linear systems resulting from continuous piecewise bilinear finite element approximations for this problem on various rectangular grids. Results reported are for uniform grids, though similar results were obtained using graded meshes. The matrix A arising from such a finite element approximation (with the usual nodal basis functions and the natural ordering of nodes) is a nine-stripped matrix, as pictured in section 2.

Table 2 shows the number of iterations required by each of the methods to achieve a relative error of size 10^{-4} times that of the relative error in the initial guess:

$$\frac{\|x^k - \bar{x}\|_2}{\|x^0 - \bar{x}\|_2} \leq 10^{-4}.$$

The "exact" solution $\bar{x}$ was obtained by iterating for many steps, until the pseudo-residual, $M^{-1}(b - A\bar{x})$, was of size 10^{-8} or less. In each case, a random initial guess was used.

Note that, as predicted by the estimates in Table 1, the number of iterations required by the DSCG and ICCG methods increases more rapidly with grid size than does that required by the other methods. Note also, however, that while the convergence rate of DSCG, ICCG, and HBDSCG is almost independent of the diffusion coefficient $\rho(x,y)$, the other methods -- HBCG and NDCG -- slow down considerably when applied to problems with discontinuous, strongly varying diffusion coefficient. This behavior cannot be predicted from the asymptotic estimates of Table 1. The degradation in performance of these methods makes them impractical for the types of problems described in [10], for example, despite the mild dependence of the convergence rate on the mesh size.

The estimates of Table 1 predict convergence of the NDCG method in $O(\log d/h)$ iterations. Asymptotically, this is correct, though the iteration counts in Table 2 seem to grow with the mesh size even when d/h is fixed. This is because the grid size is not yet fine enough to approach the asymptotic bound. For a 128x128 grid with $\rho(x,y)=1$, NDCG with $d/h=8$ required 15 iterations, just as for the 64x64 grid, indicating that this is probably close to the maximum number of iterations that would be required on any size grid. For the discontinuous diffusion coefficient problems, the size of grid necessary to approach the constant bound on iteration count is probably much finer, and the constant bound much larger than the number of iterations listed in Table 2.

$\rho(x,y)$	grid size	# Iterations					
		DSCG	ICCG	HBCG	HBDSCG	NDCG ($\frac{d}{h}=8$)	NDCG ($\frac{d}{h}=16$)
1.	16x16	21	8	14	14	6	
	32x32	43	13	18	18	11	7
	64x64	84	25	21	21	15	13
if $x < .5$, $\rho = 1$. if $x \geq .5$, $\rho = 100$.	16x16	23	8	88	15	8	
	32x32	49	14	>>100	19	29	9
	64x64	>100	28	>>100	28	41	32
if $x < .5$, $y < .5$, $\rho = 1$. if $x \geq .5$, $y < .5$, $\rho = 1.E4$ if $x < .5$, $y \geq .5$, $\rho = 1.E-4$ if $x \geq .5$, $y \geq .5$, $\rho = 1$.	16x16	24	9	>>100	12	7	
	32x32	48	15	>>100	14	32	9
	64x64	97	24	>>100	15	>100	61
$.01 + x^2 + y^2$	16x16	22	8	51	15	7	
	32x32	43	13	76	19	11	7
	64x64	84	25	>>100	23	15	13

Table 2. Number of Iterations Required with Various Grid Sizes and Different Functions $\rho(x,y)$.

For comparison, we include in Table 2b another set of tests run on the same problems, but this time with a *zero* initial guess. Note that the iteration counts in Table 2b are considerably better than those of Table 2, for all methods tested, with some methods showing more of a difference than others. This is because with a smooth (and symmetric) right-hand side and a zero initial guess, the initial residual is entirely low frequency. It is deficient in many of the eigenvectors of the iteration matrices and so one sees faster convergence than could be predicted by spectral analysis. While smoothness of the initial residual might be a reasonable assumption for some problems, we will generally report results with the random initial guess, to better illustrate the theory. It should be pointed out that many of the results reported in the literature [e.g., 6,11] use a zero initial guess with a similar right-hand side function, and this should be taken into account by the reader, since it can significantly affect iteration counts and comparisons.

$\rho(x,y)$	grid size	# Iterations					
		DSCG	ICCG	HBCG	HBDSCG	NDCG ($\frac{d}{h}=8$)	NDCG ($\frac{d}{h}=16$)
1.	16x16	10	6	9	9	2	
	32x32	19	9	11	11	8	2
	64x64	37	16	12	12	10	9
if $x < .5$, $\rho = 1$. if $x \geq .5$, $\rho = 100$.	16x16	18	6	65	10	7	
	32x32	35	10	86	11	21	8
	64x64	70	20	>> 100	13	30	25
if $x < .5$, $y < .5$, $\rho = 1$. if $x \geq .5$, $y < .5$, $\rho = 1.E4$ if $x < .5$, $y \geq .5$, $\rho = 1.E-4$ if $x \geq .5$, $y \geq .5$, $\rho = 1$.	16x16	11	8	>> 100	8	5	
	32x32	22	13	>> 100	10	28	5
	64x64	43	28	>> 100	12	69	33
$.01 + x^2 + y^2$	16x16	19	7	41	11	5	
	32x32	37	12	54	12	9	5
	64x64	74	22	59	13	11	11

Table 2b. Number of Iterations Required with Various Grid Sizes and Different Functions $\rho(x,y)$, using Zero Initial Guess.

Table 2 shows only numbers of iterations, but as seen from Table 1, some of the methods require more work per iteration than others. In Table 3, timings are reported for the various (serial) codes run on the NYU Ultracomputer Prototype [7]. The timings are for the case $p(x,y)=1$, with a 64×64 grid and a random initial guess. The time per iteration determined for this case can be used with the iteration counts of Table 2 to compute relative performance for the other problems. The time per iteration is roughly proportional to the work per iteration, given in Table 1. This is to be expected on a single processor with no vector capabilities. The comparisons of Table 3 might look significantly different, however, on a vector processor such as the Cray 1, especially if all or some sections of the codes were carefully optimized in assembly language. In particular, the DSCG timings would look better compared to ICCG because of better vectorization. Also shown in Table 3 is a comparison of the iterative methods with a direct band matrix solver, SPBFA/SL, from LINPACK [5]. It is seen that for this problem all of the iterative methods outperform the band solver on this machine though, again, the comparison might look different on a vector machine.

Timings are given for the individual parts of the codes described in section 3. The setup times consists of the time to invert the diagonal of A in the DSCG and HBDSCG codes, the time to form the incomplete Cholesky decomposition in the ICCG code, and the time to factor each D - and N - block and the Shur complement matrix corresponding to the crosspoints in the NDCG code. It is seen that most parts contribute significantly to the total time required by the various methods. Hence, if a good speedup is to be obtained through parallelization, it will be necessary to parallelize almost all sections of the codes. A modified code, ICCG', is included in Table 3. This uses the same algorithm as the original ICCG code but determines the order in which solution components can be computed to take advantage of parallelism in the MSOLVE operation (the diagonal order described in section 2), and then solves for the components in this order. (See [8].) The original ICCG code solves with the natural ordering and is slightly more efficient as a serial code. It is the ICCG' code that will be parallelized, however, and speedups reported will be the ratio of the time for ICCG' with 1 processor to its time with P processors. The reader should keep in mind that ICCG' requires a small amount of additional overhead in the setup and MSOLVE sections and so will represent an improvement over ICCG only if the MSOLVE operation parallelizes well enough to make up for this additional overhead.

section	DSCG		ICCG		ICCG'		HBDSCG		NDCG ($\frac{d}{h}=8$)		NDCG ($\frac{d}{h}=16$)	
	time	%	time	%	time	%	time	%	time	%	time	%
setup	22	0	1208	5	1912	7	22	0	2388	8	4957	12
ATIMES	23341	46	7107	27	7202	26	6011	29	9114	29	11137	28
MSOLVE	2161	4	10593	40	10804	38	8367	40	16954	54	21633	54
other (BLAS)	24814	49	7517	28	8256	29	6363	31	2801	9	2206	6
TOTAL	50338		26425		28174		20763		31257		39933	
# iters	84		25		25		21		15		13	
time/iter	599		1009		1050		988		1925		2690	

Direct Solution		
SPBFA	SPBSL	Total
65134	3897	69031

Table 3. Serial Code Timings ($p(x,y) = 1$, 64x64 grid, random initial guess).

The codes were parallelized in a very simple way. All arrays and a few frequently used variables were declared SHARED, and appropriate DO loops were replaced by DOALL/ENDALL constructs [2], so that different iterates could be executed in parallel. DOALL prespawns processes and assigns work to them when it is available. When a DOALL statement is encountered, each process takes an index from the loop and performs the work for that index then returns to the queue to grab another index. No work after the ENDALL statement can be performed until the work associated with each loop index has been completed.

Table 4 shows the parallel code timings (for the 64x64 grid, with $p(x,y)=1$, and a random initial guess) and speedups using 1, 2, 4, 6, and 8 processors. These timings are plotted in Fig. 1, with optimal speedup represented by the dashed lines. Using 8 processors, most methods obtained about a factor of 6 speedup over their execution time on a single processor. Note, however, that the time for the parallel codes run on a single processor is

somewhat greater (about 15-30% greater) than the time for the serial codes shown in Table 3. This is due to the added overhead of accessing shared variables and taking loop indices from a queue.

Only ICCG' failed to achieve nearly optimal speedup, because the MSOLVE portion of this code achieved only a factor of 2.39 speedup. All sections of the codes (except the setup in DSCG, ICCG', and HBDSCG) were parallelized, and all parts (except the MSOLVE in ICCG') obtained approximately this factor of 6 speedup. The ATIMES routine actually achieved about a factor of 6.7 speedup, enhanced by unrolling of DO (DOALL) loops. It is believed that a larger problem size would result in somewhat better speedup, up to about a factor of seven. Bus traffic on the current Ultracomputer Prototype appears to limit the speedup of codes using shared variables to about this factor of seven. There is a slight bend in the speedup curve for NDCG at $P=6$ processors. This is because the number of blocks (64) is not a multiple of the number of processors and so there is some delay due to unbalanced workloads.

# proc.	section	DSCG		ICCG'		HBDSCG		NDCG ($\frac{d}{h}=8$)	
		time	spdup	time	spdup	time	spdup	time	spdup
1	setup	22		1950		22		3463	
	ATIMES	27102		8343		7084		10761	
	MSOLVE	4881		17162		10844		18022	
	BLAS	32499		9822		8241		3652	
	Total	64504		37277		26191		35898	
2	setup	23	1	1958	1	22	1	1774	1.95
	ATIMES	13759	1.97	4203	1.99	3550	1.99	5492	1.96
	MSOLVE	2514	1.94	11484	1.49	5605	1.93	9197	1.96
	BLAS	16898	1.92	5167	1.90	4274	1.93	1895	1.93
	Total	33194	1.94	22812	1.63	13451	1.95	18358	1.96
4	setup	22	1	1950	1	22	1	945	3.66
	ATIMES	7223	3.75	2166	3.85	1845	3.84	2899	3.71
	MSOLVE	1291	3.78	7858	2.18	2989	3.63	4884	3.69
	BLAS	9086	3.58	2762	3.56	2292	3.60	1058	3.45
	Total	17622	3.66	14736	2.53	7148	3.66	9786	3.67
6	setup	22	1	1942	1	22	1	728	4.76
	ATIMES	5026	5.39	1537	5.43	1305	5.43	2403	4.49
	MSOLVE	948	5.15	7225	2.38	2215	4.90	3713	4.85
	BLAS	6750	4.81	2040	4.81	1676	4.92	785	4.65
	Total	12746	5.06	12746	2.92	5218	5.02	7629	4.71
8	setup	22	1	1949	1	22	1	551	6.28
	ATIMES	4046	6.70	1248	6.69	1056	6.71	1692	6.36
	MSOLVE	745	6.55	7185	2.39	1777	6.10	2368	6.28
	BLAS	5769	5.63	1733	5.67	1381	5.97	651	5.61
	Total	10582	6.10	12115	3.08	4236	6.18	5762	6.23

Table 4. Parallel Code Timings ($p(x,y) = 1.$, 64x64 grid, random initial guess).

It was stated in section 3 that for n large enough ($\sqrt{n}/2 \gg P$), the ICCG' code should also achieve almost a full factor of P speedup. This was based on a theoretical argument, however, estimating the amount of parallelism present, and it did not take into account the overhead for accessing queues and checking on completion of DOALL loops. The larger the problem size n , the greater the number of DOALL loops in the MSOLVE routine, there being one for each of the approximately $3\sqrt{n}$ diagonals of the grid. Additionally, more total queue accesses are required to grab the total $O(n)$ loop indices. It is reasonable to assume that the overhead associated with executing and terminating DOALL's in the MSOLVE routine using P processors is of the form:

$$O_P = a_P \sqrt{n} + b_P n, \quad a_P, b_P \text{ independent of } n.$$

If this overhead is neglected, then the time to perform the MSOLVE operation in ICCG' can be estimated based on the amount of work in each stage of the computation. Define a unit of time to be the time required to solve a single equation of the sparse triangular linear system (using one processor), and assume that $\sqrt{n}/2$ is a multiple of the number of processors P . Following the diagonal pattern of the grid shown in section 3, one finds as in [8] that the time required to perform the backsolve using P processors is

$$\begin{aligned} t_P &= 2P*1 + 2P*2 + \dots + 2P * \frac{\sqrt{n}/2}{P} + (\sqrt{n} + 2(P-1)) * \frac{\sqrt{n}/2}{P} + \\ &\quad 2P * \left(\frac{\sqrt{n}/2}{P} - 1 \right) + \dots + 2P*1 \\ &= \frac{n}{P} + \sqrt{n} \left(1 - \frac{1}{P} \right). \end{aligned}$$

With one processor the time is

$$t_1 = n,$$

and so the speedup is given by

$$\frac{t_1}{t_P} = P * \frac{1}{1 + \frac{P-1}{\sqrt{n}}}.$$

Combining the expressions for t_1 and t_8 with the expressions for the overhead O_1 and O_8 , we can express the measured values $\hat{t}_1$ and $\hat{t}_8$ in the form

$$\hat{t}_1 = \alpha_1 \sqrt{n} + \beta_1 n,$$

$$\hat{t}_8 = \alpha_8 \sqrt{n} + \beta_8 n.$$

Measuring $\hat{t}_1$ and $\hat{t}_8$ for $\sqrt{n} = 48, 64, 96$, and 128 , we were able to estimate the parameters $\alpha_1, \beta_1, \alpha_8$, and β_8 , and thereby determine the size of problem necessary to give a speedup in the ICCG' MSOLVE routine comparable to that seen in the other algorithms for

$\sqrt{n}=64$. The parameters determined by a least squares fit to the data were

$$\alpha_1 = 1.7261, \quad \beta_1 = .1381, \quad \alpha_8 = 3.1932, \quad \beta_8 = .0205 .$$

Plotted in Figure 2 are the observed and estimated speedups in this routine for problem sizes ranging up to $\sqrt{n}=800$. Due to memory limitations on the ultracomputer prototype, the largest problem we were actually able to solve was $\sqrt{n}=128$. If the model is correct, and if the coefficients $\alpha_1, \beta_1, \alpha_8, \beta_8$ computed from our data are accurate, then this implies that a problem size of approximately $\sqrt{n}=1154$ would be needed to obtain a factor of 6 speedup in this routine, and a factor of 7 speedup with 8 processors would not be possible. Thus, in practice, it may not be possible to attain close to a full factor of P speedup with P processors, because the overhead grows with the problem size. It should be pointed out that these estimate of problem size needed to achieve a certain speedup are extremely sensitive to the parameters $\alpha_1, \beta_1, \alpha_8, \beta_8$ obtained from measured timings. Hence they should be regarded only as rough estimates.

5. Conclusions.

Results of these experiments show the hierarchical basis method with diagonal scaling (HBDSCG) to be particularly efficient on both serial and parallel architectures. It should also be compared with other multigrid methods (such as the multigrid V-cycle), and comparisons should be made on vector machines as well.

The problems presented here are all relatively easy to solve on current serial processors. The real payoff from parallelism is likely to come for larger 3-D problems run on machines with huge memories and many more processors. One might conjecture that if eight processors can be used effectively on a 2-D problem, then eight hundred could be used on a 3-D problem consisting of one hundred 2-D pieces. Of course, this is an oversimplification, especially in the case of HBDSCG, since its convergence rate does not appear as good in 3-D as in 2-D [16]. To test the effectiveness of these algorithms on problems of real interest, we need machines with more processors and very large memories.

6. References:

- [1] F. L. Bauer, "Optimally Scaled Matrices," *Num. Math.* 5 (1963), 73-87.
- [2] W. Berke, "A Structured Environment for Parallel Fortran," Ultracomputer System Software Note, in preparation.
- [3] P. Bjorstad and O. Widlund, "Iterative Methods for the Solution of Elliptic Problems on Regions Partitioned into Substructures," *SIAM J. Num. Anal.* 23 (1986), 1097-1120.
- [4] P. Concus, G. H. Golub, and D. P. O'Leary, "A Generalized Conjugate Gradient Method for the Numerical Solution of Elliptic Partial Differential Equations," in *Sparse Matrix Computations*, ed. Bunch and Rose, Academic Press, New York (1976), 309-332.

- [5] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia (1979).
- [6] M. Dryja, W. Proskurowski, and O. Widlund, "A Method of Domain Decomposition with Cross Points for Elliptic Finite Element Problems", in *Optimal Algorithms*, Sofia, (1986), 97-111.
- [7] A. Gottlieb, "An Overview of the NYU Ultracomputer Project," Ultracomputer Note #100, Courant Institute, July, 1986.
- [8] A. Greenbaum, "Solving Sparse Triangular Linear Systems Using Fortran with Parallel Extensions on the NYU Ultracomputer Prototype," Ultracomputer Note #99, Courant Institute, April, 1986.
- [9] M. Hestenes and E. Stiefel, "Methods of Conjugate Gradients for Solving Linear Systems," *J. Research NBS* 49 (1952), 409-436.
- [10] D. S. Kershaw, "The Incomplete Cholesky Conjugate Gradient Method for the Iterative Solution of Systems of Linear Equations," *J. Comp. Phys.* 26 (1978), 43-65.
- [11] D. E. Keyes and W. D. Gropp, "A Comparison of Domain Decomposition Techniques for Elliptic Partial Differential Equations and Their Parallel Implementation," *SIAM J. Sci. Stat. Comput.*, 8 (1987), s166-s202.
- [12] J. A. Meijerink and H. A. Van der Vorst, "An Iterative Solution Method for Linear Systems of which the Coefficient Matrix is a Symmetric M-Matrix," *Math. of Comp.* (1977), 148-162.
- [13] S. H. Sameh and R. P. Brent, "Solving Triangular Systems on a Parallel Computer," *SIAM J. Numer. Anal.* 14 (1979), 1101-1113.
- [14] O. Widlund, "Iterative Substructuring Methods: the General Elliptic Case," Technical Report #260, Courant Institute, November, 1986.
- [15] O. Widlund, "Iterative Substructuring Methods: Algorithms and Theory for Elliptic Problems in the Plane," Technical Report #287, Courant Institute, March, 1987. To appear in proceedings of the First International Conference on Domain Decomposition Methods, Paris, 1987.
- [16] H. Yserentant, "On the Multi-level Splitting of Finite Element Spaces," *Num. Math.* 49 (1986), 379-412.

Fig. 1. Parallel Code Timings (64x64 grid, $p(x,y)=1.$, random initial guess).

Fig. 2. Measured (x) and predicted speedup of the MSOLVE routine in ICCG' using 8 processors for different problem sizes $\sqrt{n}$.